Fixed-Point Arithmetic 2

2.1 OVERVIEW

Binary number representations usually include a sign and a radix point, as well as a magnitude. The sign shows whether the number is positive or negative. The radix point separates the integer and fractional parts of the number.

The sign of a binary number can be represented with one bit. In most representations, a zero indicates positive and a one indicates negative. The sign bit is usually in the leftmost location (most significant bit).

There are several formats for representing negative numbers, including signed-magnitude, ones complement, and twos complement. The most common method, and the one used by the ADSP-2100, is twos complement. The advantage of twos-complement format is that it provides a unique representation for zero, whereas the other formats have both a positive and a negative zero. In twos-complement format, zero is considered positive; therefore, the magnitude of the largest negative number that can be represented with a given number of bits is one greater than the magnitude of the largest positive number. A twos-complement number of k+1 bits (one bit indicates the sign and k bits indicate the magnitude) can represent the range of numbers from $2^k - 1$ to -2^k.

The twos complement of a binary number can be calculated in one of two ways: 1) invert all the bits and add one to the least significant bit, or 2) invert all bits to the left of the least significant 1. For example:

Binary +72 0100 1000
Invert bits 1011 0111
Add 1 +
0000 0001
Binary –72 1011 1000

or

Binary +72 0100 1000
Invert all *bits* left of |
least significant 1 invert
Binary –72 1011 1000

2 Fixed-Point Arithmetic

A radix point is placed between two bits in a number. The bits to the left of the radix point represent the integer part of the number; the bits to the right of the radix point represent the fractional part of the number. There are two ways *to specify the lo*cation of the radix point:

- Fixed-point format places the radix point at a single, predetermined location. Often this location is to the left of all bits (all bits are fractional) or to the right of all bits (all bits are integer). Because the location of the radix point is assumed by software, it does not need to be represented explicitly. Arithmetic operations (such as multiplication) can change the radix-point position so that shifting may be necessary to kee*p the number in the s*ame fixed-point format.

- Floating-point format uses two numbers to represent a value: a mantissa and an exponent. The exponent indicates the location of the radix point. The exponent may be stored along with the mantissa or in a separate register.

The ADSP-2100 represents numbers in a fixed-point format. In this publication, the location of the radix point is given by the format designation I.Q, in which I is the number of bits to the left of the radix point and Q is the number of bits to the right. For example, the 1.15 format indicates signed full fractional numbers; one integer bit indicates the sign, and 15 fractional bits indicate the fractional magnitude. Full integer number representation is 16.0 format. For most signal processing applications, fractional numbers (1.15 format) are assumed. The multiplier and divider of the ADSP-2100 are optimized for use with this format.

ADSP-2100 addition, subtraction, and multiplication primitives operate directly on single-precision (16-bit) numbers. In this chapter, we show an example of how to program the ADSP-2100 to perform single-precision, fixed-point division. We also include explanations of extended-precision, fixed-point arithmetic. Example implementations of addition, subtraction, and multiplication are shown in both double precision (32-bit operands) and triple precision (48-bit operands). Division is shown using a 64-bit dividend and a 32-bit divisor.

Double-precision operations are most common, so we show examples that can be implemented directly. Triple-precision arithmetic is also shown to demonstrate how to handle the middle words of extended-precision numbers. Repeating the middle-word operations allows extension to any

precision.

2.2 SINGLE-PRECISION FIXED-POINT DIVISION

The ADSP-2100 instruction set includes two divide primitives, DIVS and DIVQ, to compute fixed-point division. The DIVS instruction calculates the sign bit of the quotient, and the DIVQ calculates a single bit of the quotient. These instructions can be used to implement a nonrestoring (remainder is invalid) add/subtract division algorithm that works with signed or unsigned operands. The operands must be either both signed or both unsigned. Because each instruction produces one bit of the quotient, dividing a 16-bit divisor into a 32-bit dividend to produce a 16-bit quotient requires 16 instructions, and therefore 16 cycles. Block diagrams of the DIVS and DIVQ operations are shown in Figures 2.1 and 2.2, on the following page.

The division algorithm performs either an addition or subtraction based on the signs of the divisor and the partial remainder. Mano, 1982, gives an excellent explanation of a similar algorithm.

In the ADSP-2100 implementation of the division algorithm for signed operands, the divisor can be stored in AX0, AX1, or any register on the R bus. The MSW of the dividend can be loaded into AY1 or AF, and the dividend's LSW is loaded into AY0. To calculate the quotient, the ADSP-2100 first executes a DIVS instruction to compute the sign of the quotient, followed by 15 DIVQ instructions to compute 15 quotient bits. A signed fixed-point division routine is shown in Listing 2.1. This routine takes the divisor from AX0, the dividend's MSW from AF, and the dividend's LSW from AY0. The quotient is returned in AY0.

In unsigned division, the dividend's MSW must be loaded into AF, and the ASTAT register must be cleared to set the AQ bit to zero. The ADSP-2100 executes 16 DIVQ instructions. Listing 2.2 shows a subroutine to perform an unsigned division. The registers must be preloaded with the same values as for the signed division routine: the divisor in AX0, the dividend's MSW in AF, and the dividend's LSW in AY0.

The format of the quotient is determined by the format of the two operands. If the dividend is in P.Q format, and the divisor is in M.N format, the quotient will be in (P–M+1).(Q–N–1) format. Some format manipulation may be necessary to guarantee the validity of the quotient. For example, if both operands are signed and fully fractional (dividend in 1.31 format and divisor in 1.15 format) then the result is fully fractional (in 1.15 format), and therefore the dividend must be smaller than the divisor for a valid quotient.

2 Fixed-Point Arithmetic

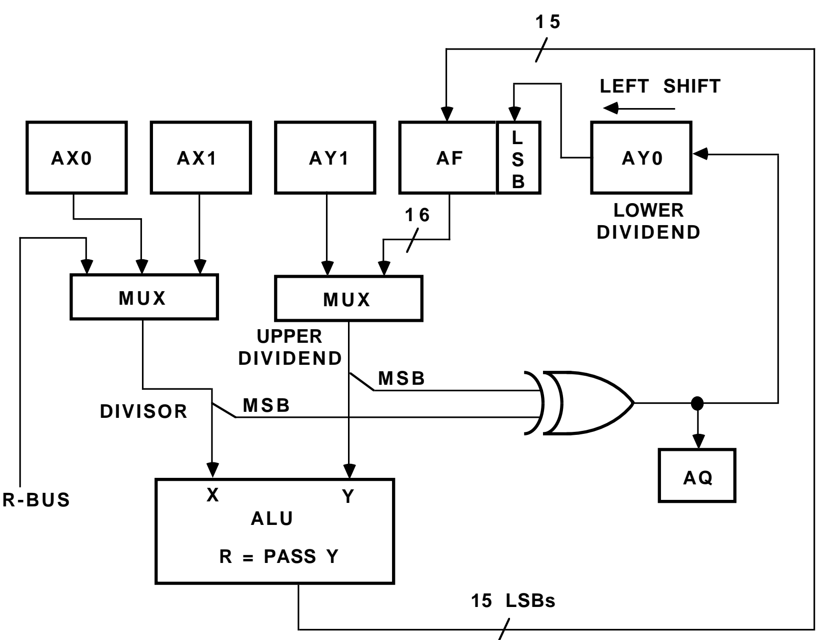

Figure 2.1 DIVS Block Diagram

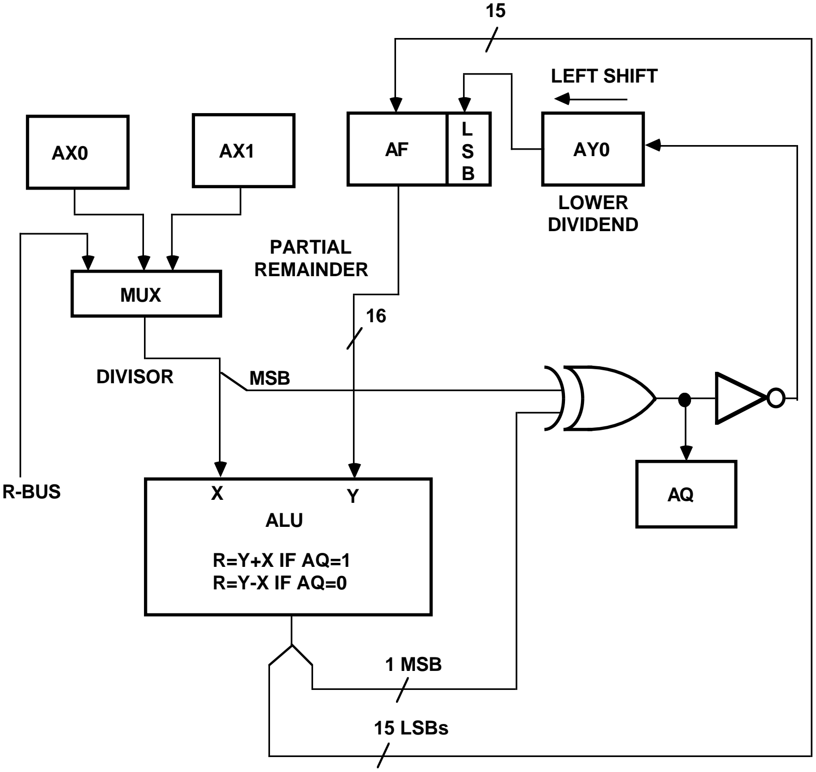

Figure 2.2 DIVQ Block Diagram

To divide two integers (dividend in 32.0 format and divisor in 16.0 format) and produce an integer quotient (in 16.0 format), you must shift the dividend one bit to the left (into 31.1 format) before dividing.

```
.MODULE Signed_SP_Divide;

{
   Signed Single-Precision Divide

   Calling Parameters
        AF = MSW of dividend
        AY0 = LSW of dividend
        AX0 = 16-bit divisor

   Return Values
        AY0 = 16-bit result

   Altered Registers
        AY0, AF

   Computation Time
        17 cycles

}
.ENTRY  sdivs;

sdivs:  DIVS AF,AX0;                            {Compute sign bit}
        DIVQ AX0; DIVQ AX0; DIVQ AX0;           {Compute 15 quotient bits}
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        RTS;
.ENDMOD;
```

Listing 2.1 Single-Precision Divide, Signed

```
.MODULE Unsigned_SP_Divide;

{
   Unsigned Single-Precision Divide

   Calling Parameters
        AF = MSW of dividend
        AY0 = LSW of dividend
        AX0 = 16-bit divisor

   Return Values
        AY0 = 16-bit result

   Altered Registers
        AY0, AF

   Computation Time
        18 cycles

}

.ENTRY  sdivq;

sdivq:  ASTAT=0;                       {Clear AQ bit of ASTAT}
        DIVQ AX0;                                     {Compute 16
quotient bits}
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        RTS;
.ENDMOD;
```

Listing 2.2 Single-Precision Divide, Unsigned

2.3 MULTIPRECISION FIXED-POINT ADDITION

The following algorithm adds two multiprecision operands together:

1. Add the two LSWs to produce the LSW of the result and a carry bit.
2. Add the next word of each operand plus the carry from the previous word to produce the next word of the result and a carry bit.
3. Repeat step 2 until every word of the result has been computed. After the MSW has been computed, the status flags of the ALU will be valid

for the multiprecision sum.

To produce a valid number, the radix points must be in the same location in both operands. The result will be in the same format as the operands.

Listing 2.3 shows a subroutine that implements the addition algorithm for two double-precision numbers. The LSW of the augend is stored in AX0, and its MSW is stored in AX1. The LSW of the addend is stored in AY0, and its MSW is stored in AY1. The LSW of the result is returned in SR0, and its MSW is returned in SR1.

Listing 2.4 shows a subroutine that performs triple-precision addition. This routine retrieves the augend from data memory, LSW first, starting at DM(I0). The addend is read from data memory, LSW first, starting at DM(I1). The result is stored in data memory, LSW first, starting at DM(I2). Each data memory access modifies the address by adding the value of M0 to the I register used in the access. Before executing the routine, you must ensure that the augend and addend are loaded at the correct data memory locations, and you must initialize I0, I1, I2, and M0. You must also set the buffer length registers L0, L1, and L2 to zero to disable circular buffers and allow the ADSP-2100 to read the full numbers. Note that most of the subroutine instructions are concerned with either reading operands or writing the result.

```
.MODULE Double_Precision_Add;

{  Double-Precision Addition
        Z = X + Y

   Calling Parameters
        AX0 = LSW of X
        AX1 = MSW of X
        AY0 = LSW of Y
        AY1 = MSW of Y

   Return Values
        SR0 = LSW of Z
        SR1 = MSW of Z

   Altered Registers
        AR,SR

   Computation Time
```

(listing continues on next page)

LISTING 2.4 TRIPLE-PRECISION ADDITION

2.4 MULTIPRECISION FIXED-POINT SUBTRACTION
The subtraction algorithm is very similar to the addition algorithm. In fact, subtraction can be accomplished by adding the twos complement of the subtrahend to the minuend. Multiprecision subtraction is performed by the following steps:

1. Subtract the LSW of the subtrahend from the LSW of the minuend to produce the LSW of the result and a borrow (“carry–1”) bit.
2. Subtract the next word of the subtrahend from the next word of the minuend and add to it the “carry – 1,” producing the next word of the result and a carry bit. The “carry – 1” value effectively implements a borrow signal from the previous word.
3. Repeating step 2 until every word of the result has been computed. After the MSW has been computed, the status flags of the ALU will be valid.

Listing 2.5 shows a double-precision subtraction routine. This routine assumes that the minuend’s LSW is stored in AX0, the minuend’s MSW is in AX1, the subtrahend’s LSW is in AY0, and the subtrahend’s MSW is in AY1. The result is returned in the SR registers (LSW in SR0, MSW in SR1).

Listing 2.6 shows the triple-precision subtraction routine. Most of its instructions perform data I/O. The minuend is read, LSW first, starting at DM(I0); the subtrahend is read, LSW first, starting at DM(I1). The result is written, LSW first, starting at DM(I2). Before calling this routine, you must initialize I0, I1, I2 to the correct data memory locations, M0 to one (the memory spacing value) and L0, L1, and L2 to zero (to disable circular buffers).

```
.MODULE Double_Precision_Subtract;

{  Double-Precision Subtraction
         Z = X - Y

   Calling Parameters
         AX0 = LSW of X
         AX1 = MSW of X
         AY0 = LSW of Y
         AY1 = MSW of Y
```

(listing continues on next page)

```
   Return Values
        SR0 = LSW of Z
        SR1 = MSW of Z

   Altered Registers
        AR, SR

   Computation Time
        4 cycles
}

.ENTRY  dps;

dps:    AR=AX0-AY0;                 {Subtract LSWs}
        SR0=AR, AR=AX1-AY1+C-1;     {Subtract MSWs}
        SR1=AR;
        RTS;
.ENDMOD;
```

Listing 2.5 Double-Precision subtraction

```
.MODULE Triple_Precision_Subtract;

{  Triple-Precision Subtraction
        Z = X - Y

   Calling Parameters
        I0 —> X Buffer              L0 = 0
        I1 —> Y Buffer              L1 = 0
        I2 —> Z Buffer              L2 = 0
        M0 = 1

   Return Values
        Z Buffer is filled

   Altered Registers
        I0, I1, I2, AR, AX0, AY0

   Computation Time
        10 cycles
}

.ENTRY  tps;

tps:    AX0=DM(I0,M0);                    {Fetch LSWs}
        AY0=DM(I1,M0);
        AX0=DM(I0,M0), AR=AX0-AY0;        {Subtract LSWs}
        AY0=DM(I1,M0);
        DM(I2,M0)=AR, AR=AX0-AY0+C-1;     {Store LSW, fetch middle}
                                          {words}
```

```
        AX0=DM(I0,M0);                {Fetch MSWs}
        AY0=DM(I1,M0);
        DM(I2,M0)=AR, AR=AX0-AY0+C-1;   {Subtract MSWs}
        DM(I2,M0)=AR;                 {Store MSW}
        RTS;
.ENDMOD;
```

LISTING 2.6 TRIPLE-PRECISION SUBTRACTION

2.5 MULTIPRECISION FIXED-POINT MULTIPLICATION

Multiplication is more complicated than either addition or subtraction. An important task is placing the radix point in the product. In addition and subtraction, the radix point location in the result is the same as for both operands. In multiplication, the radix point may be in a different location in the two operands, and its location in the product depends on the locations in the two operands. The result of multiplying a number in P.Q format by a number in M.N format produces a result in (P+M–1).(Q+N+1) format. The ADSP-2100 multiplier automatically shifts the product one bit to the left to eliminate the redundant sign bit of the result. Therefore, multiplication of two numbers in a full fractional format (1.15 format, the most common format) returns the result in a full fractional format (1.31). If two numbers in full integer format (16.0) are multiplied, the result is in 31.1 format. To produce an integer result (32.0 format), the product must be shifted to the right one bit before storing it in memory.

Multiplication is performed according to the following procedure, which is illustrated in Figure 2.3, on the next page.

1. Multiply each word of the multiplicand by the LSW of the multiplier and an appropriate power of two to shift it left to its correct position.
2. Multiply each word of the multiplicand by the next word of the multiplier and the appropriate power of two.
3. Repeat step 2 until each word of the multiplicand has been multiplied by every word of the multiplier.

U and N are 16-bit words

$$
\begin{array}{r}
N_n\,N_{n-1}\,...\,N_2\,N_1\,N_0 \\
\times \qquad U_n\,U_{n-1}\,...\,U_2\,U_1\,U_0 \\
\hline
2^{16n}U_0N_n + 2^{16(n-1)}U_0N_{n-1} + ...2^{32}U_0N_2 + 2^{16}U_0N_1 + U_0N_0 \\
2^{16(n+1)}U_1N_n + 2^{16n}U_1N_{n-1} + ... \quad 2^{48}U_1N_2 + 2^{32}U_1N_1 + 2^{16}U_1N_0 \\
\vdots \\
+ 2^{16(2n)}U_nN_n + ...2^{16(n+1)}U_nN_1 + 2^{16n}U_nN_0
\end{array}
$$

Figure 2.3 Multiprecision Multiplication

4. Add all the partial products together.

In computing the partial products, the signed/unsigned switch of the multiplier determines whether the multiplication is signed, mixed-mode, or unsigned. Multiplication of the MSWs should be signed, and multiplication of a less significant word by either MSW should be mixed-mode. All other multiplication should be unsigned.

Listing 2.7 shows a double-precision multiplication routine for fractional operands. The routine assumes that the multiplicand's LSW is stored in MX0 and its MSW is stored in MX1, and that the multiplier's LSW is stored in MY0 and its MSW is stored in MY1. This routine produces a 64-bit product that is stored in data memory, LSW first, starting at DM(I0). M0 and L0 should be set to one and zero, respectively, before the routine is executed.

The product of the LSWs of the operands is computed first, and the LSW of the result is written to DM(I0). The MR register is then shifted 16 bits to the right. The inner products are computed and added to the shifted value in MR; MR0 is written to the next data memory location. The MR register is again shifted 16 bits to the right, and the product of the MSWs is computed and added to MR. MR0 is written to data memory, followed by MR1, to complete the 64-bit product. Note how the signed/unsigned switch indicates the type of multiplication for each partial product.

The triple-precision multiplication routine is shown in Listing 2.8. In this

routine, the multiplicand and the multiplier are both stored in data memory, LSW first, the multiplicand starting at DM(I0) and the multiplier starting at DM(I1). The routine produces a 96-bit result stored LSW first, starting at DM(I2). The X and Y buffers must be declared (and located in memory) as circular buffers; L0 and L1 are set to three because the routine circles back to refetch the first words of the operands. Before executing the routine, you should set M0 to one and L2 to zero.

Listings 2.9 and 2.10 show the double-precision routine and the triple-precision routine, respectively, for integer multiplication. These routines differ from the multiplication routines already described only in that they shift the result one bit to the right before writing it to memory, in order to generate a full integer product.

```
.MODULE Double_Precision_Multiπly;

{  Double-Precision Multiplication
        Z = X ¥ Y

   Calling Parameters
        I0 —> Address of Z Buffer   L0 = 0
        M0 = 1
        MX0 = LSW of X
        Mx1 = MSW of X
        MY0 = LSW of Y
        MY1 = MSW of Y

   Return Values
        Z Buffer Filled

   Altered Registers
        MR, I0

   Computation Time
        13 cycles

}

.ENTRY  dpm;

dpm:    MR=MX0*MY0(UU);             {Compute LSW}
        DM(I0,M0)=MR0;              {Save LSW}
        MR0=MR1;                    {Shift right 16 bits}
        MR1=MR2;
```

```
        MR=MR+MX1*MY0(SU);           {Compute inner product}
        MR=MR+MX0*MY1(US);
        DM(I0,M0)=MR0;               {Shift right 16 bits}
        MR0=MR1;
        MR1=MR2;
        MR=MR+MX1*MY1(SS);           {Compute MSW}
        DM(I0,M0)=MR0;
        DM(I0,M0)=MR1;               {Store MSW}
        RTS;
.ENDMOD;
```

Listing 2.7 Double-Precision Multiplication

```
.MODULE Triple_Precision_Multiπly;

{  Triple-Precision Multiplication
        Z = X ¥ Y

   Calling Parameters
        I0 —> X Buffer              L0 = 3
        I1 —> Y Buffer              L1 = 3
        I2 —> Z Buffer              L2 = 0
        M0 = 1

   Return Values
        Z Buffer Filled

   Altered Registers
        MX1,MX0,MY1,MY0,MR,I0,I1,I2

   Computation Time
        26 cycles
}

.ENTRY  tpm;

tpm:    MY0=DM(I0,M0);
        MX0=DM(I1,M0);
        MX1=DM(I1,M0), MR=MX0*MY0(UU);     {Compute LSW}
        DM(I2,M0)=MR0;                     {Save LSW}
        MR0=MR1;                           {Shift right 16 bits
        MR1=MR2;
        MY1=DM(I0,M0), MR=MR+MX1*MY0(UU);
        MR=MR+MX0*MY1(UU);
        DM(I2,M0)=MR0;
        MR0=MR1;                           {Shift right 16 bits}
        MR1=MR2;
        MY1=DM(I0,M0), MR=MR+MX1*MY1(UU);
        MX0=DM(I1,M0), MR=MR+MX0*MY1(US);
        MY0=DM(I0,M0), MR=MR+MX0*MY0(SU); {Skip 1st word, LSW}
```

```
        DM(I2,M0)=MR0;
        MR0=MR1;                         {Shift right 16 bits}
        MR1=MR2;
        MY0=DM(I0,M0), MR=MR+MX1*MY1(US);
        MR=MR+MX0*MY0(SU);
        DM(I2,M0)=MR0;
        MR0=MR1;
        MR1=MR2;
        MR=MR+MX0*MY1(SS);
        DM(I2,M0)=MR0;
        DM(I2,M0)=MR1;                           {Save MSW}
        RTS;
.ENDMOD;
```

Listing 2.8 Triple-Precision Multiplication

```
.MODULE Integer_DPM;

{
   Integer Double-Precision Multiplication
        Z = X ¥ Y

   Calling Parameters
        I0 —> Z Buffer                 L0 = 0
        M0 = 1
        MX0 = LSW of X
        MX1 = MSW of X
        MY0 = LSW of Y
        MY1 = MSW of Y
        SE = -1

   Return Values
        Z Buffer Filled

   Altered Registers
        I0, MR, SR

   Computation Time
        14 cycles

}

.ENTRY  idpm;

idpm:   MR=MX0*MY0(UU);                  {Compute LSW}
```

```
        MR0=MR1, SR=LSHIFT MR0(LO);       {Shift LSW right 1 bit}
        MR1=MR2, SR=SR OR LSHIFT MR1(HI); {before saving}
        DM(I0,M0)=SR0, MR=MR+MX1*MY0(SU);
        MR=MR+MX0*MY1(US);
        MR0=MR1, SR=LSHIFT MR0(LO);
        MR1=MR2, SR=SR OR LSHIFT MR1(HI);
        DM(I0,M0)=SR0, MR=MR+MX1*MY1(SS);
        SR=LSHIFT MR0(LO);
        SR=SR OR LSHIFT MR1(HI);
        SR=SR OR LSHIFT MR2 BY 15(HI);
        DM(I0,M0)=SR0;
        DM(I0,M0)=SR1;                    {Save MSW after
shifting}
        RTS;
.ENDMOD;
```

Listing 2.9 Integer Double-Precision Multiplication

```
.MODULE Integer_TPM;

{  Integer Triple-Precision Multiplication
        Z = X ¥ Y

   Calling Parameters
        I0 —> X Buffer                L0 = 3
        I1 —> Y Buffer                L1 = 3
        I2 —> Storage for Z           L2 = 0
        M0 = 1
        SE = -1

   Return Values
        Z Buffer Filled

   Altered Registers
        MX0,MX1,MY0,MY1,MR,I0,I1,I2,SR

   Computation Time
        28 cycles
}

.ENTRY  itpm;

itpm:   MY0=DM(I0,M0);                    {Fetch LSWs}
        MX0=DM(I1,M0);
        MY1=DM(I0,M0), MR=MX0*MY0(UU);    {Compute LSW}
        MX1=DM(I1,M0);
        MR0=MR1, SR=LSHIFT MR0(LO);       {Shift LSW}
        MR1=MR2, SR=SR OR LSHIFT MR1(HI);
        DM(I2,M0)=SR0, MR=MR+MX0*MY1(UU); {Store LSW}
        MR=MR+MX1*MY0(UU);
        MR0=MR1, SR=LSHIFT MR0(LO);
        MR1=MR2, SR=SR OR LSHIFT MR1(HI);
        DM(I2,M0)=SR0, MR=MR+MX1*MY1(UU);
        MY1=DM(I0,M0);
```

```
        MR=MR+MX0*MY1(US), MX0=DM(I1,M0);
        MY0=DM(I0,M0), MR=MR+MX0*MY0(SU); {Skip 1st word}
        MR0=MR1, SR=LSHIFT MR0(LO);
        MR1=MR2, SR=SR OR LSHIFT MR1(HI);
        DM(I2,M0)=SR0;
        MY0=DM(I0,M0), MR=MR+MX1*MY1(US);
        MR=MR+MX0*MY0(SU);
        MR0=MR1, SR=LSHIFT MR0(LO);
        MR1=MR2, SR=SR OR LSHIFT MR1(HI);
        DM(I2,M0)=SR0, MR=MR+MX0*MY1(SS);
        SR=LSHIFT MR0(LO);                {Shift MSW}
        SR=SR OR LSHIFT MR1(HI);
        SR=SR OR LSHIFT MR2 BY 15(HI);
        DM(I2,M0)=SR0;
        DM(I2,M0)=SR1;                    {Save MSW}
        RTS;
.ENDMOD;
```

Listing 2.10 Integer Triple-Precision Multiplication

2.6 MULTIPRECISION FIXED-POINT DIVISION

The routine shown in Listing 2.11 provides a convenient method for doing double-precision division (64-bit dividend, 32-bit divisor). It is a double-precision software implementation of the algorithm used by the hardware instructions DIVS and DIVQ (see section 2.2). Mano, 1982, gives an excellent explanation of a similar algorithm. The division algorithm generates one bit of the quotient by comparing the divisor and the partial remainder. If the divisor is less than or equal to the partial remainder, the quotient bit is a one; otherwise, the quotient bit is a zero. The divisor is subtracted from the partial remainder if it is less than the partial remainder, and then the divisor is shifted right one bit and compared to the partial remainder to generate the next bit. This routine shifts the dividend to the left rather than the divisor to the right, accomplishing the same comparison.

Before calling the routine, you must load SE with –15. You must also load the dividend into the SR and MR registers. Its MSW should be loaded into SR1, the next word in SR0, the next in MR1, and its LSW in MR0. The divisor should be loaded into the AY registers, LSW in AY0 and MSW in AY1. The result is returned in the MR registers, LSW in MR0 and MSW in

MR1.

The division subroutine has two entry points; ddivs sho*uld b*e called to execute a signed division, and ddivq should be called to execute an unsigned divis*ion.* The section of the routine starting at the ddivs label calculates the sign bit by comparing the MSW of the divisor with the MSW of the dividend and shifts the dividend one bit left. The CNTR register is set to 31, the number of bits that remain to be calcula*ted.* The section of the routine starting at the ddivq label sets the CNTR register to 32 and AX1, which is used to compare the divisor and the partial remainder, to zero. The portion of the routine com*mon t*o both signed and unsigned division is the ddivu loop. In this loop, the divisor is subtracted from the partial remainder if both have the same sign; otherwise, the divisor is added to the partial remainder. The quotient bit is determined by comparing the MSW of the divisor (in AX0) with the partial remainder (in AF). The dividend is shifted one bit left, and the loop is repeated until all 32 bits of the quotient have been computed.

```
.MODULE     Double_Precision_Divide;

{
      Double-Precision Division
            Z = X ÷ Y

      Calling Parameters
            AY0 = LSW of Y
            AY1 = MSW of Y
            SR1 = MSW of X
            SR0 = Next Significant Word of X
            MR1 = Next Significant Word of X
            MR0 = LSW of X
            SE = -15

      Return Values
            MR1 = MSW of Z
            MR0 = LSW of Z

      Altered Registers
            AF,AR,AX1,AX0,SI,SR,MR
```

```
      Computation Time
            485 cycles (maximum)
}

.ENTRY      ddivs;
.ENTRY      ddivq;

ddivs:      AF=PASS SR1;
            SI=SR0, AR=SR1 XOR AY1;   {Exclusive OR sign bits}
            AX1=AR;
            SR=LSHIFT MR0 BY 1(LO);   {shift dividend up 1 bit}
            SR=SR OR LSHIFT MR1 BY 1(HI);
            SR=SR OR LSHIFT AR(LO);   {shift in quotient-sign bit}
            AR=PASS AF, MR0=SR0;
            MR1=SR1, SR=LSHIFT MR1(LO);
            SR=SR OR LSHIFT SI BY 1(LO);
            SR=SR OR LSHIFT AR BY 1(HI);
            CNTR=31;
            JUMP ddiv;
ddivq:      CNTR=32;
            AX1=0;
ddiv:       AX0=AY1;
            DO ddivu UNTIL CE;
                  AR=ABS AX1;                {is quotient bit set?}


                  IF POS JUMP aqz;           {no, -divisor from
partial remainder}
aqo:              AR=SR0+AY0;                {yes, +divisor to
partial remainder}
                  SI=AR, AF=SR1+AY1+C;
                  JUMP ddivi;
aqz:              AR=SR0-AY0;
                  SI=AR, AF=SR1-AY1+C-1;
ddivi:            SR=LSHIFT MR0 BY 1(LO);    {shift dividend 1
bit}
                  SR=SR OR LSHIFT MR1 BY 1(HI);
                  AR=AX0 XOR AF;             {compute quotient
bit}
                  AX1=AR;                    {save quotient
bit}
                  AR=NOT AX1;
                  SR=SR OR LSHIFT AR(LO);    {shift in new bit}
                  MR0=SR0, AR=PASS AF;
                  MR1=SR1, SR=LSHIFT MR1(LO);
                  SR=SR OR LSHIFT SI BY 1(LO);
ddivu:            SR=SR OR LSHIFT AR BY 1(HI);
                  RTS;
.ENDMOD;
```


LISTING 2.11 DOUBLE-PRECISION *DIVISION*